Information and Software Technology 102 (2018) 6–29

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Debugging and maintaining pragmatically reused test suites

Soha Makady[a], Robert J. Walker*,[b]

[a] Computer Science Department, Faculty of Computers and Information, Cairo University, Giza, Egypt
[b] Department of Computer Science, University of Calgary, Calgary, Canada

ARTICLE INFO

Keywords:
Pragmatic software reuse
Test suite reuse
Error detection
Error repair
Debugging
Maintenance
Semi-automatic
Lightweight tool

ABSTRACT

Context: Pragmatic software reuse is a common activity in industry, involving the reuse of software artifacts not designed to anticipate that reuse.
Objective: There are two key issues in such tasks that have not been previously explored. (1) Subtle bugs can be inserted due to mistakes on the part of a developer performing the pragmatic reuse. The reused code, integrated in the target system, should be (re-)validated there. But it is not clear what validation strategies would be employed by professional developers, and which of these strategies would be most effective to detect and to repair these inserted bugs. (2) Although semi-automated reuse of the associated test suite has been previously proposed as a strategy to detect such inserted bugs, it is unknown if the reused test suite would be maintainable in practice and how its maintenance characteristics would compare against alternative strategies.
Method: We present two empirical studies with industrial developers to address these open issues.
Results: We find that industrial developers use a few strategies including test suite reuse, but that test suite reuse is more reliably effective at discovering and repairing bugs inserted during pragmatic reuse. We also find that, in general, semi-automatically reused test suites are slightly more maintainable than manually reused test suites, in pragmatic reuse scenarios; specific situations can vary wildly however. Participants suggested specific extensions to tool support for semi-automated reuse of test suites.
Conclusions: While various validation strategies are employed by industrial developers in the context of pragmatic reuse, none is as reliable and effective as test case reuse at discovering and repairing bugs inserted during pragmatic reuse. Despite the fact that semi-automatically reused test cases contain non-trivial adaptive code, their maintainability is equivalent to or exceeds that of manually reused test suites. The approach could be improved, however, by adopting the suggestions of our participants to increase usability.

1. Introduction

Software reuse encourages the development of new software systems by leveraging existing artifacts. Reuse has long been promoted for its potential to increase the productivity of software developers, to reduce development time, and to decrease defect density [5,7,55,72]. Most research into software reuse has focused on pre-planned approaches, such as object-oriented inheritance [13,39], software components [55,74], and software product lines [45,63]. Unfortunately, pre-planned reuse has drawbacks: (1) prediction is difficult as to what artifacts should be built for reuse [77]; (2) it is too expensive to build all artifacts for reuse [4,11]; and (3) artifacts cannot be reused intact in arbitrary contexts because of their embedded assumptions [25,49].

Instead, software developers sometimes find themselves in situations where existing artifacts do not quite meet their needs. Rather than reimplementing the functionality of interest or refactoring the software where the functionality exists, developers often perform an ad hoc but *pragmatic* process of copy-and-modify on portions of the existing source code [31].[1] Pragmatic reuse is known to be an industrially common practice [6,11,33,36,40,44,68,78,80], and it *can* be the disciplined action of a developer who has carefully weighed the risks involved [31,80]. Nevertheless, pragmatic reuse could cause subtle bugs when constraints that are met within the originating system are violated in the target system [33].

To prevent silent introduction of bugs during pragmatic reuse, the developer has three known options: (1) use automated test generation techniques [e.g., 12, [22,28,61], 83]; (2) create new automated test suites (e.g., via JUnit or other xUnit family members[2] [56]); or

* Corresponding author.
E-mail addresses: s.makady@fci-cu.edu.eg (S. Makady), walker@ucalgary.ca, walker@lsmr.org (R.J. Walker).

[1] Pragmatic reuse has been known by a variety of other terms: copy-and-modify reuse [46], code scavenging [44], ad hoc reuse [64], software salvaging [9], opportunistic reuse [68], unanticipated reuse [30,76], software transplantation [29], and clone-and-own [17], although some subtle differences exist about the context of application.

[2] https://www.martinfowler.com/bliki/Xunit.html [accessed 10 November 2017].

https://doi.org/10.1016/j.infsof.2018.05.001
Received 31 March 2017; Received in revised form 20 February 2018; Accepted 5 May 2018
Available online 19 May 2018
0950-5849/

(3) pragmatically reuse and adapt test suites from the originating system. Note that automated test generation techniques require detailed knowledge of expected behavior of the system (as opposed to actual behavior), by means of: detailed formal specifications (which are unlikely to exist in industrial settings); manually supplied program invariants (which require expertise with the reused code that developers performing pragmatic reuse tasks lack [8,31,43,50]); or feedback about whether actual behavior is correct or not (which again requires expertise with the reused code). Furthermore, new automated test suites are expensive to create manually [58,75] and the developer's superficial understanding of the reused code would cause such new test suites to be of questionable value. Apparently, reusing and adapting the original test suite is the best option. Currently, we have no evidence about two questions: (RQ1) What strategies would developers use to discover and repair errors inserted during pragmatic reuse? (RQ2) What strategies are most effective to discover and repair errors inserted during pragmatic reuse?

We conduct a semi-controlled experiment to address these two questions. We discover a set of alternative approaches to validate pragmatic reuse tasks, and we compare the merits of developers' chosen approaches against reuse and adaptation of the original test suite. Our results show that, while developers do attempt a variety of other strategies, identification and repair of errors is more successful when test suite adaptation and reuse is pursued.

Given the value of reuse and adaptation of the original test suite, we previously considered how this process can be automated [51,52], addressing the problem of leveraging the original test suite to validate pragmatically reused functionality. We semi-automatically reuse the portions of a test suite that are associated with the reused functionality. This permits faults to be detected that were introduced during pragmatic reuse, minimizing false alarms. The approach is reified in a tool called Skipper that uses a record-and-replay (R&R) technique to partially transform the originating system's unit tests to only exercise the reused code, placing them in the target system: runtime information is serialized during an execution of the test suite on the originating system, and deserialized for use within the execution of the transformed test suite on the target system.

Although we demonstrated that Skipper was more effective than an alternative manual reuse approach [52], two additional open questions remain. (RQ3) Are Skipper's R&R tests harder to maintain than manually written tests? (RQ4) How would developers validate pragmatically reused code in the absence of R&R tests?

To address these questions, we performed a case study into the maintainability of Skipper's R&R test suites in the presence of various disruptive changes caused by the evolution of reused source code. The developers maintained manually-written or Skipper R&R tests, after we had applied various test-breaking changes to different portions of some reused source code; the changes were mostly behavior-modifying, requiring non-trivial understanding of the reused code to repair broken tests. We then interviewed the developers about the pros and cons of both approaches, and how they would test such reused code in the absence of Skipper. Our results indicate that developers can successfully maintain R&R tests; they would struggle with the same kinds of missing information as in manually created tests. Furthermore, developers see R&R tests as appropriate where creating manual tests is too difficult, particularly for complex or unfamiliar reused functionality.

The paper is structured as follows. Section 2 describes a running example in which a developer must validate pragmatically reused code. Section 3 details a semi-controlled experiment into validation strategies employed by professional developers during pragmatic reuse, addressing RQ1 and RQ2. Section 4 overviews background of our previous work on Skipper's R&R test suites. Section 5 continues our running example, to demonstrate potential problems of maintaining Skipper's R&R test suites. Section 6 describes our case study and interviews into the maintainability of Skipper R&R test suites in the presence of various disruptive changes, addressing RQ3 and RQ4. Section 7 discusses remaining issues. Section 8 describes related work.

2. Motivation: validating pragmatically reused features

Consider a scenario in which a developer is building a new application, which we refer to as "YouTube Recommender", to recommend musical videos to users based on their musical taste. For instance, if a user prefers to listen to pop songs specifically, that application would recommend to him any YouTube videos of songs from that same genre. The developer happens to know of aTunes [18,19], a system for managing and playing audio files. Among its various features, aTunes provides the *related artists* feature, similar to the one desired. When an aTunes user plays a song for some artist, aTunes recommends a set of related artists to which this user might also want to listen. He decides to reuse that feature within his new application.

2.1. Performing pragmatic reuse

The developer explores the source code of aTunes (the originating system) to identify the starting point of the source code responsible for the *related artists* feature. He settles on the public method `getSimilarArtists()`, located in the `LastFmService` class. Accordingly, he copies the source code for that method into his target project, which results in dangling references to other classes and methods. He proceeds to add and modify other relevant source code elements from within aTunes, needed for the related artists feature to work properly within his new context.

The developer copies a total of 22 source classes from the aTunes project into the organization's (new) target project, making various modifications to the copied code to keep only those portions that serve the *related artists* feature. For instance, the `LastFMService` class originally included 2 inner classes, 20 fields, and 39 methods spanning 640 non-comment LOCs within aTunes. After closely exploring the `getSimilarArtists()` method, the developer decides to keep all of its source code (a small sample of this code is shown in Fig. 1), except for one line performing some logging functionality that he is not interested in. Within the target YouTube Recommender system, the reused class `LastFMService` includes only 18 fields and 3 methods, spanning 116 non-comment LOCs.

2.2. Validating the pragmatically reused code

Within YouTube Recommender, the developer wants the *related artists* feature to retain the same behavior it possessed within aTunes. But he is unable to obtain the desired recommendations from it. After a frustrating and lengthy debugging session, he determines that the `getSimilarArtists()` always returns a blank list of recommendations, despite the requisite initialization having been performed correctly. Evidently, some error was introduced during the pragmatic reuse of the feature.

The problem is rooted in the fact that the resource file `ehcache-lastfm.xml` was missed during the pragmatic reuse of the *related artists* feature, thereby damaging its functionality. That resource file is used to correctly initialize the `lastFmCache` field within the `LastFMService` class (line 2 in Fig. 1). However, as that resource file is not explicitly mentioned within the `LastFMService` class, the developer failed to copy it to the target system during the initial reuse, in which he focused on investigating specifically the source code. This error results from the developers in the reusing organization having limited knowledge about the aTunes project's structure and configuration.

Not knowing these details, the developer is unclear about what has gone wrong. His manual debugging has not helped him to localize the fault within the reused feature. He could leverage the wealth of knowledge stored within aTunes system's test suite, by also pragmatically reusing the original, automated test suite for the *related artists*